JFP **14** (2): 211–245, March 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796803004751 Printed in the United Kingdom

Constructed product result analysis for Haskell

CLEM BAKER-FINCH
Department of Computer Science, The Australian National University, Canberra ACT 0200, Australia
(*e-mail:* clem@cs.anu.edu.au)

KEVIN GLYNN
Department of Computer Science, The University of Melbourne, Melbourne, Australia
(*e-mail:* keving@cs.mu.oz.au)

SIMON PEYTON JONES
Microsoft Research, Cambridge, UK
(*e-mail:* simonpj@microsoft.com)

Abstract

Compilers for ML and Haskell typically go to a good deal of trouble to arrange that multiple arguments can be passed efficiently to a procedure. For some reason, less effort seems to be invested in ensuring that multiple results can also be returned efficiently. In the context of the lazy functional language Haskell, we describe an analysis, *Constructed Product Result* (CPR) analysis, that determines when a function can profitably return multiple results in registers. The analysis is based only on a function's definition, and not on its uses (so separate compilation is easily supported) and the results of the analysis can be expressed by a transformation of the function definition alone. We discuss a variety of design issues that were addressed in our implementation, and give measurements of the effectiveness of our approach across a substantial benchmark set. Overall, the price/performance ratio is good: the benefits are modest in general (though occasionally dramatic), but the costs in both complexity and compile time, are low.

1 Introduction

A good compiler for ML or Haskell will ensure that multiple arguments are passed to a function using an efficient calling convention. For example, consider a function of type

```
f :: (Int, Int) -> Int
```

This function takes a pair of `Ints` as its (single) argument. While the programmer *may* apply `f` to an existing pair, it is much more common for the argument pair to be constructed at the call site; that is, a call often looks like `f (e1, e2)`. In this case, a good compiler can simply pass the two arguments in registers, rather than boxing them into a pair first; this transformation is called *argument flattening* (Appel, 1992), or *arity raising* (Hannan & Hicks, 1998).[1]

[1] Scheme finesses the problem by making multi-argument functions part of the language.

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

It is also common for functions to return multiple results:

```
g :: Int -> (Int, Int)
```

There is no reason in principle why returning multiple results should be any less efficient than passing multiple arguments – after all, the same registers are available – but in practice it is less often optimised. In this paper, we present an analysis and accompanying program transformation for Haskell that optimises functions that return multiple results. Our contributions are as follows:

- We show how to use the so-called 'worker/wrapper transformation' to optimise the calling convention of a function that returns a product (section 2).
- We have implemented the analysis and transformation in the Glasgow Haskell Compiler (GHC), a state-of-the-art compiler for Haskell. As is usually the case, a large set of issues were exposed as we scaled up the original idea into a real implementation, as we discuss in section 3.
- We present an analysis to determine when it would be helpful for a function to return its result unboxed (section 4). This analysis is, by design, not type-based: it is based on the actual form of the function definition.
- We give measurements to show the effectiveness of the transformation in practice (section 6).

Our approach has the following desirable properties:

- It is simple to describe and implement.
- It is compatible with separate compilation (section 5.2).
- It never increases the amount of heap allocation.
- It is safe for space (section 3.11).

Other approaches either cannot offer these performance guarantees, or else require a rather sophisticated whole-program analysis. We discuss related work in section 7.

2 The idea

We begin by outlining the main idea of the paper. Consider the following function:

```
dm :: Int -> Int -> (Int, Int)
dm x y = (x `div` y, x `mod` y)
```

We would like to express the fact that `dm` can return its result pair unboxed. We do this by splitting `dm` into a *wrapper* and a *worker*, thus:

```
dm :: Int  -> Int -> (Int,  Int)
dm x y = case dmw x y of
           (# r1, r2 #) -> (r1, r2)
```

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

```
dmw :: Int -> Int -> (# Int, Int #)
dmw  x  y = (# x  `div`  y,  x  `mod` y #)
```

The worker, dmw, returns an *unboxed tuple*; we write unboxed tuples using '(#' and '#)' parentheses in both types and terms. The idea, of course, is that the calling convention for a function that returns an unboxed tuple arranges to return the components in registers. Unboxed tuples are a built-in type in our compiler (Peyton Jones & Launchbury, 1991).

The wrapper, dm, serves as an impedance matcher to interface callers expecting to call dm with the worker dmw. The wrapper is inlined at every call site. So, given the call

```
case dm x y of
  (p,q) -> <e>
```

we inline dm, to obtain

```
case (case dmw x y of
        (# r1, r2 #) -> (r1,r2)) of
  (p,q) ->  <e>
```

which can readily be transformed to

```
case dmw x y of
  (# p,q #) -> <e>
```

In this case, the result of dm is immediately scrutinised by a case expression, and indeed this is quite common. But if not, nothing is lost. We simply end up with a call looking like this:

```
...(case dmw x y of (# r1,r2 #) -> (r1,r2))...
```

All that has happened is that the code required to allocate the result pair has moved from the body of dm to the call site.

2.1 *The general worker/wrapper transform*

The example function dm had a right-hand side that consisted solely of a constructor application, so the worker/wrapper split was easy to perform. But if the body of the function is not a simple constructor application, it is less obvious how to construct the body of the worker.

However, the following transformation does the worker/wrapper split regardless of the form of the original function. Suppose that the definition of f is:

```
f :: Int -> (Int, Int)
f x = <e>
```

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

for some arbitrary expression "<e>". Then the following transformation is always valid:

```
f :: Int -> (Int, Int)
f x = case fw x of
        (# r1,r2 #) -> (r1,r2)

fw :: Int -> (# Int, Int #)
fw x = case <e>  of
         (r1,r2) -> (# r1,r2 #)
```

Indeed, if this general transformation is applied to the function dm described earlier, the body of dmw quickly simplifies to the form given.

As we shall see later, pairs and general tuples are examples of *product types* (single-constructor algebraic data types) and our analysis extends smoothly to all such types.

2.2 Returning single arguments unboxed

So far we have stressed the benefits of returning multiple arguments efficiently. But the same transformation can be extremely beneficial even when returning a *single* result, because instead of returning the result wrapped in a heap-allocated box, we can return the contents of the box. A particularly important special case is the Int type. In GHC, the Int type is defined like this:

```
data Int = MkInt Int#
```

Here, Int# is the type of 32-bit unboxed integers (Peyton Jones & Launchbury, 1991); a value of type Int is represented by a heap-allocated MkInt constructor whose contents is of type Int#.

Functions over Int are defined using pattern matching, as with any other algebraic data type. For example, a function that increments its argument can be expressed like this:

```
inc :: Int -> Int
inc (MkInt x) = MkInt (x +# 1)
```

where +# is the function (actually, machine instruction) that adds two unboxed integers. Unless we do something about it, this function returns a boxed Int in the heap, just as dm returns a boxed pair. The worker-wrapper transformation gives:

```
inc :: Int -> Int
inc x = case (incw x) of
            y -> MkInt y

incw :: Int -> Int#
incw (MkInt x) = x +# 1
```

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

(The slightly strange-looking phrase "case (incw x) of y -> ..." means "evaluate incw x and bind the result to y"; using case rather than let expresses the eagerness of the call.) The worker function incw returns its result, of type Int# in a register, rather than heap-allocating a MkInt box as the original function inc did. In numerically-intensive code avoiding these allocations can be a very big win, as we show in section 6.

2.3 When is worker/wrapper beneficial?

Although the worker/wrapper transformation we have described is *correct*, it is not necessarily *beneficial*. In particular, the worker fw takes apart and discards the pair returned by <e>, only for the wrapper f to reconstruct it. *This actually makes things worse, unless the* case *expression in* fw *is certain to cancel with the construction of the pair in* <e>, as is certainly the situation for our example dm.

Consider the function hdPr:

```
hdPr :: [(Int,Int)] -> (Int,Int)
hdPr (x:xs) = x
```

Here, we do not want to return the pair unboxed, because the call site might need the pair in boxed form. If we unbox uniformly, we risk repeatedly re-constructing a pair that already exists as a member of the list. Furthermore, unboxing it in hdPr gains nothing: the pair already exists, so no allocation is saved.

The bottom line is this: we should only perform the CPR worker/wrapper transformation if the result of the function is an *explicitly-constructed* product; that is, one allocated by the function itself. We say that such functions *have the CPR property* – CPR stands for 'constructed product result'. This contrasts with a type-based analysis, simply driven by the type of the function[2], and which would therefore treat dm and hdPr uniformly.

So we cannot just apply the worker/wrapper transformation to any function that returns a product type. Will a simple syntactic inspection of the function body do instead? Alas no. What if f looks like this?

```
f x = let ... in
      case ... of
        p1 -> ( <e1> , <e2>  )
        p2 -> ( <e3> , <e3>  )
```

Clearly, f does indeed allocate its return pair: we need to 'look inside' let and case expressions. What if we have a function g which tail-calls another function?

```
g x = h x x
h a b = (b,a)
```

That is, g does not construct its result explicitly, but h does. If we do a worker/wrapper split on h, and inline the wrapper in g, we get

[2] More precisely we mean 'driven by the *existing* program types'. The abstract interpretation we describe next could of course be formulated as a type system!

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

```
g x = case hw x x of
        (# r1,r2 #) -> (r1,r2)
```

So now g *does* explicitly construct its result. That is, in the end, g has the CPR property if h does. If we had anticipated this, we could have done the worker/wrapper split on g as well as h.

In short, we need to perform an analysis to identify those functions for which the worker/wrapper split is desirable. A simple abstract interpretation will do the job nicely, as we describe in section 4.

2.4 Strictness analysis and argument flattening

For many years we have been using another sort of worker/wrapper transformation to express unboxing of function arguments, based on *strictness* information (Peyton Jones & Launchbury, 1991; Peyton Jones & Santos, 1998). For example, if `f :: (Int,Int) -> Int` is strict, then we split f to

```
f x = case x of
        (a,b) -> fw a b
fw a b = let x = (a,b) in <rhs>
```

where `<rhs>` is the original right-hand side of f. This is, of course, simply the traditional argument-flattening transformation, but in a lazy language argument-flattening is only valid for strict functions.

We do not discuss the details here, but simply note that it is easy to make a single worker/wrapper transformation that deals simultaneously with unboxing both arguments and results, thus taking advantage of the information from both strictness analysis and CPR analysis. For example, the inc function from Section 2.2 is certainly strict, so by combining strictness and CPR information GHC generates the following worker/wrapper split:

```
inc :: Int -> Int
inc x = case x of
          MkInt x' -> case (incw x') of
                             y -> MkInt y

incw :: Int# -> Int#
incw x = x +# 1
```

Now consider the following calls of inc:

```
inc (inc a)
```

Without optimisation, the inner call to inc allocates a MkInt box for its result, which is immediately taken apart by the outer inc. However, if we inline the wrapper for inc twice, and simplify, the MkInt from the inner call cancels with the case from the outer call, so we end up with:

```
case a of
   MkInt a' -> MkInt (incw (incw a'))
```

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

The intermediate result is no longer allocated.

2.5 Summary

The unifying idea of our compiler is that of *compilation by transformation* (Peyton Jones & Santos, 1998), in which the program is transformed by a succession of correctness-preserving steps into a more efficient form. Many of these transformations depend on the ability to expose in the intermediate language some more concrete aspects of the target implementation. In particular, our intermediate language includes unboxed types (such as unboxed integers and tuples).

Within this transformational framework, CPR analysis works as follows:

- First, we perform a large number of generic transformations (Peyton Jones & Santos, 1998), including inlining, collectively implemented by the *simplifier*.
- Next, we do strictness analysis and CPR analysis.
- Based on the information from both these analyses, we perform the worker/wrapper transformation.
- Now we re-apply the simplifier, which will inline the wrappers at each call site and use the extra detail at each call site to further simplify calls to our desired form.

The worker/wrapper transformation is entirely local. The non-local effects rely on a fairly aggressive inliner which, in effect, propagates the local effects of the worker/wrapper transformation to the non-local call sites (Peyton Jones & Marlow, 1998). As other transformations take effect new call sites may show up, so the wrapper may be inlined long after the worker/wrapper transformation is performed.

The whole idea of the worker/wrapper transformation is that it moves just a little of the function's computation from the function definition to its call sites. However, if the body of the original function is very small, as is the case for `dm` above, then it is probably better instead to inline the *entire* function. In our compiler we disable the worker/wrapper split for functions with small bodies, but our examples in this paper will be small for the purposes of presentation. The worker/wrapper approach to exposing a function's optimised calling protocol was independently invented by Goubault (1994), who called it *partial inlining*.

3 Working out the details

The basic idea of our analysis is simple but, as is often the case, the exercise of implementing it in a production compiler showed up a number of unforeseen details. In this section we describe these interactions. This material is a core contribution of the paper: it bridges the gap between a promising idea and a real implementation.

3.1 User-defined products

So far we have focused on functions that return tuples and `Int`s. What about functions that return values of other types?

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

A *product* type is any algebraic data type that has exactly one constructor. The built-in tuple types are certainly products, but the programmer may define new ones; for example:

```
data Point = MkP Int Int
diag :: Int -> Point
diag  x = MkP x x
```

The same worker/wrapper transformation works for any function, such as `diag`, that returns a product. We do not need a new type of unboxed points for the worker to return: an unboxed tuple of suitable arity will do fine.

```
diag x = case diagw x of
            (# a,b #) -> MkP a b
diagw x = (# x,x #)
```

In GHC, unboxed tuples are not first-class citizens. The exist only to encode the return-in-registers return convention, and are allowed only to the right of a function arrow. Curiously enough, unboxed 1-tuples are sometimes required. Consider the following product type:

```
data Age = MkAge Int

age :: Age -> Age
age a = case a of
          Age n -> Age (n+1)
```

The function is certainly strict, and certainly has the CPR property. Here is the correct worker/wrapper split:

```
age :: Age -> Age
age a = case a of
          Age n -> case agew n of
                        (# n1 #) -> Age n1

agew :: Int -> (# Int #)
agew n = (# n+1 #)
```

Notice that the worker returns a *singleton* unboxed tuple. It would be wrong to write:

```
agew :: Int -> Int
agew n = n+1           -- WRONG
```

because then `agew` would be strict in `n`, whereas the original function is not. Operationally, a singleton unboxed tuple says “stop evaluating and return the result now”.

We have seen that GHC uses an ordinary algebraic data type, albeit with an unboxed component, to represent `Int` values (section 2.2). GHC uses similar algebraic data types to represent characters and floating-point numbers (both single

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

and double precision). Such data types are usually built in, but they are not treated specially by GHC. Since they are all single-constructor types, they are all products, and can be handled by the same analyses and transformations as user-defined data types.

In the case of `incw` (section 2.2), however, the worker could safely return the single result *without wrapping it in an unboxed 1-tuple*, because an unboxed `Int#` is already fully evaluated. There is no difference in runtime representation between returning `x` and returning `(# x #)`, where `x` has type `Int#`, but the former seems a little more economical. So GHC's general rule is this: the worker of a CPR function returns its result(s) in an unboxed tuple, unless there is exactly one result of unlifted type (such as `Int#`, `Float#` etc).

3.2 Recursive types and sum types

So far we have spoken only about functions that return a value of a product type (i.e. an algebraic data type with one constructor). What of recursive types, and sum types? A recursive type is no problem, provided it is a product: CPR analysis simply unboxes the top level of the result value.

However, if the type has more than one constructor, it is less obvious how to return it unboxed. In principle one could imagine returning a tag, indicating the constructor that is being returned, and zero or more other fields. To express this idea as a source-to-source transformation we would have to introduce unboxed sum types. This is entirely possible, and the payoff might be considerable – lists are extremely common – but it would require some extra machinery in the intermediate language for unboxed sums, and we leave that for future work.

Returning an enumerated value, e.g. a `Bool`, does not require any allocation because the returned value will be a pointer to a pre-existing constant in the heap. But saving allocation is not everything, unboxing can still produce benefits. If, as is commonly the case, the caller dispatches on the value returned by the call, it would be preferable to return an unboxed value: the dispatch code would then save a memory fetch. However, in this paper we restrict our attention solely to product types.

3.3 Recursive functions

Consider the following function:

```
g :: Int -> (Int,Int)
g x = if x < 0 then (1,1) else g (x-1)
```

This function has the CPR property, because when (and if) it returns, it will return a constructed product. The CPR analysis takes the form of an abstract interpretation that calculates the abstract value of `g` in the standard way, using a sequence of approximations, starting from $\perp$. To make `g` have the CPR property, we should therefore treat the abstract value $\perp$ as having the CPR property. In the extreme, a function that always diverges has the CPR property vacuously – it never returns.

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

(Similarly, strictness analysis regards a function that always diverges as strict; if g $\perp = \perp$ we can soundly use call-by-value even if g diverges without evaluating its argument.)

Having decided that g does have the CPR property, the worker/wrapper transform gives:

```
g :: Int -> (Int,Int)
g x = case gw x of
        (# p,q #) -> (p,q)
gw :: Int -> (# Int,Int #)
gw x = case (if x < 0 then (1,1) else g (x-1)) of
         (p,q) -> (# p,q #)
```

The definition of gw rapidly simplifies to

```
gw x = if x < 0 then (# 1,1 #)
       else case g (x-1) of
              (p,q) -> (# p,q #)
```

Recall that the wrapper, g, should be inlined at *all* its call sites, *including any call sites in the worker*. If we inline g inside gw, and then simplify, the case expression in the body of gw becomes:

```
case (case gw (x-1) of (# p,q #) -> (p,q)) of
   (p,q) -> (# p,q #)
```

Interchanging the case expressions (a transformation GHC does a great deal) gives:

```
case gw (x-1) of
    (# p,q #) -  case (p,q) of
                   (p,q) -  (# p,q #)
```

Now the inner case cancels out:

```
case gw (x-1) of
    (# p,q #) -> (# p,q #)
```

A final routine transformation gives:

```
gw (x-1)
```

So the simplified definition of gw becomes:

```
gw x = if x < 0 then (# 1,1 #) else gw (x-1)
```

This is just the result we hoped for: the worker is a simple, tight loop, while the wrapper serves as an impedance-matcher for external callers.

3.4 Dealing with exceptions

Consider the following function:

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

```
chr :: Int -> Char
chr (MkInt i) = if (i> =0 && i<=255) then MkChar i
                else error "Bad arg to chr"
```

The function `error` prints a message and brings execution to a halt.[3] Semantically, its value is just $\perp$.

Does `chr` have the CPR property? Clearly, yes! If it returns at all, it certainly returns a constructed product, and that is all that matters. So when performing CPR analysis, we should treat a call to `error` (which is a built-in function) as having the CPR property. This is entirely compatible with the treatment of recursion in the previous section, where we agreed that the abstract value $\perp$ has the CPR property. Once this is done, any 'dressed up' versions of `error` will automatically also have abstract value $\perp$, and hence have the CPR property, as well. For example:

```
panic :: String -> a
panic msg = error ("Panic: " ++ msg)
```

Taking advantage of error values in this way turns out to be important in practice. A significant minority of functions have an error case that is expected never to occur, and it is very galling if these error cases unnecessarily prevent the CPR transformation from happening.

3.5 Conditionals and constants

What should happen if one arm of a conditional constructs a product, but the other arm does not? In principle, the Right Thing depends on the relative frequency with which the arms are taken, but our current strategy is to be pessimistic: if either arm fails to return a constructed product we deem that the conditional does not either.

An important special case concerns constants. Consider the factorial function:

```
one = MkInt 1

fact :: Int -> Int
fact n = if n==0 then one else n * fact (n-1)
```

Does `fact` have the CPR property? Well, '`*`' does, so the `else` branch does. But what about the constant `one`? Since `one` is bound directly to a manifest constructor, we may imagine inlining `one` at this usage site (without duplicating any work), and hence we deem that it *does* have the CPR property despite our comments in the previous section. Since many recursive functions have constant base cases, this is a good pragmatic choice. The inlining of `one` happens automatically, elsewhere in the compiler.

While this choice usually works well, we have found one or two programs for which it actually *increases* allocation. Consider:

[3] In a more refined semantics, `error` raises an exception (Peyton Jones *et al.*, 1999), but everything we say here remains valid.

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

```
signum :: Int -> Int
signum n = if      n > 0  then one
           else if n==0 then zero
           else              minus_one
```

where `zero` and `minus_one` are defined similarly to `one`. `signum` returns *only* constants, so if a caller happens to only want the result boxed, the wrapper for `signum` will box the result at each call rather than sharing the boxed values `one`, `zero`, etc. Here is a place where it might be helpful to have two versions of the function: one that returned its results boxed, and one that returned them unboxed. We do not support multiple versions at present. Instead we accept that the CPR transformation can occasionally increase allocation, and experiments confirm that the effect is indeed rare (section 6).

3.6 Nested and higher-order functions

Many functions are defined locally within some other function definition – not at the top level of the program – and some may have the CPR property. Our analysis has no difficulty with such definitions. We present measurements of how often such definitions actually occur in practice in section 6.

Our system works in the context of a higher-order language, but if a function is passed as an argument to another it must use the standard, boxed, return convention. So the more that higher-order functions are inlined, the more opportunities we are likely to find for returning unboxed results.

3.7 Preserving laziness

The worker/wrapper transformation must be careful to preserve laziness. Consider:

```
f :: Int -> (Int,Int)
f = let x = <expensive>  in \y -> (x,y)
```

Does `f` have the CPR property? It certainly evaluates to a function that returns a constructed product. But it would be a Bad Idea to perform the worker/wrapper transform to:

```
f y = case fw y of
        (# p,q #) -> (p,q)
fw y = let x = <expensive>  in (# x,y #)
```

The new version of `f` gives the same result but the trouble now is that under call-by-need the expression `<expensive>` is evaluated afresh for each application of `f`, whereas in the original formulation it would be shared among all calls to `f`. It follows that we should analyse a function for the CPR property only if its lambdas are ‘manifest’; that is, if the definition is of the form

```
f = \y -> <body>
```

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

More precisely, we insist that there must be enough lambdas to satisfy all the arrows in `f`'s type.

For a similar reason we also insist that there must be at least one lambda. For example, consider:

```
t = if <e>  then (y,z) else (z,y)
```

It looks as though `t` has the CPR property, but it would be wrong to blindly apply the worker/wrapper transformation to `t`, because that would give:

```
t :: (Int, Int)
t = case tw of (# p,q #) -> (p,q)

tw :: (# Int,Int #)
tw = if <e>  then (# y,z #) else (# z,y #)
```

In the original definition, `t` might never be evaluated, and hence `<e>` might not either. But in the new program `tw` is an unboxed value, so its evaluation cannot be delayed – delaying implies boxing (Peyton Jones & Launchbury, 1991). So `tw` would have to be evaluated strictly, which might change the meaning of the program.

To summarise: we can be sure to preserve laziness if we only perform CPR analysis on definitions *(a) that are of functional type* and *(b) all of whose lambdas are syntactically manifest*. Under these conditions, the worker/wrapper transformation is unconditionally *sound*; the purpose of the analysis is only to (conservatively) approximate whether the transformation will be *beneficial*.

We still analyse the body of any function disqualified by these conditions, in case it contains local function definitions that have the CPR property.

3.8 Join points

When performing the so-called 'case-of-case' transformation, GHC often introduces 'join points' (Peyton Jones & Santos, 1998). For example, if the Haskell programmer writes

```
f x y = if (x && y) then <e1>  else <e2>
```

GHC will (after inlining `&&` and doing a few transformations) derive:

```
f x y = let j = <e2>
        in case x of
             False -> j
             True  -> case y of
                        False -> j
                        True  -> <e1>
```

Here, `j` is a 'join point' where execution joins if either `x` or `y` is `False`. Now suppose that both `<e1>` and `<e2>` have type `(Int,Int)`, and both return a constructed pair; that is, both have the CPR property. Then we would like the whole expression to have the CPR property. But we appear to be stymied, because `j` is a thunk, and hence won't be CPR'd because of the considerations of Section 3.7.

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

Join points are very special, though: *a join point is entered at most once*. (Once for each instantiation of the `let` that binds the join point, that is.) It would be possible to record this information somehow, and make use of it in the CPR analyser, but a simpler alternative is to turn each join point into a function by giving it a dummy argument, thus:

```
f x y = let j = \v -> <e2>
        in case x of
             False -> j void
             True  -> case y of
                        False -> j void
                        True  -> <e1>
```

`j` is now a function of one dummy argument, of type `Void`;[4] since it is used at most once, no sharing is lost by this transformation. Because it is now a function, the CPR analyser can analyse it, find that it has the CPR property, and so `f` in turn will have the CPR property. For example, if `<e1>` is `(x,y)` and `<e2>` was `(y,x)`, the resulting code will be

```
f x y = case fw x y of
          (# p,q #) -> (p,q)

fw x y = let jw = \v -> (# y,x #)
         in case x of
              True -> jw void
              False -> case y of
                         False -> jw void
                         True  -> (# x,y #)
```

Since GHC introduces join points at a single point during its optimisation passes, it is easy to introduce the dummy argument where necessary (Peyton Jones & Santos, 1998).

3.9 Exploiting strictness

Preserving laziness is important, but we run CPR analysis just after strictness analysis. Can we exploit that strictness information to improve CPR results? It did not even occur to us to ask this question until we started to look the results of CPR analysis on real programs, but it turns out that there are two distinct settings in which strictness analysis can be a real help.

The first relates to non-recursive `let` bindings. Consider the binding for `t` from the previous section, but this time in a context that is strict in `t`:

[4] A ML programmer would use the empty tuple, `()`, here; but in Haskell the type `()` has two members, namely `()` and $\perp$. `Void` is a built-in type in GHC that has only one value, $\perp$, written `void`. Having only one value, we do not generate any code to pass a value of type `Void`. So there is no cost for the dummy argument.

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

```
let
    t = if <e>  then (y,z) else (z,y)
in
<body>
```

where <body> is guaranteed (by strictness analysis) to evaluate t. Here, t's right hand side certainly has the CPR property, so we can transform the let into a case with a pair pattern:

```
case (if <e>  then (y,z) else (z,y)) of
    (p,q) -> let t = (p,q)
             in <body>
```

This is only going to be beneficial if we can make the new case cancel with the constructed products (y,z) and (z,y). Since <body> might be large, GHC builds a join point (abstracted over p and q) before doing the case-of-case transformation (the details of abstracted join points are discussed in Peyton Jones & Santos (1998)):

```
let j p q = let t = (p,q) in <body>
in
if <e>  then
    case (y,z) of (p,q) -> j p q
else
    case (z,y) of (p,q) -> j p q
```

Now the cases cancel to give:

```
let j p q = let t = (p,q) in <body>
in
if <e>  then j y z
       else j z y
```

Notice that t is bound to a plain pair, which has the CPR property without any danger of lost sharing, and that in turn might improve the results for the enclosing function. To summarise, *we may safely treat a non-recursive* let *binder as having the CPR property if (a) its right hand side does, and (b) the binder is guaranteed to be evaluated by the body of the* let.

There is a second way in which we can exploit strictness information. Consider the tail-recursive factorial function:

```
fac n m = if n==0 then m
                  else fac (n-1) (m*n)
```

Does it have the CPR property? Apparently not, because it returns the accumulating parameter, m. *But the strictness analyser will discover that* fac *is strict in* m, *so it will be passed unboxed to the worker for* fac. More concretely, here is the worker/wrapper split that will result from strictness analysis alone:

```
fac n m = case n of MkInt n' ->
          case m of MkInt m' ->
          facw n' m'
```

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

```
facw n' m' = if n' ==# 0# then MkInt m'
                                else facw (n' -# 1#) (m' *# n')
```

Looking at this, it is clear that facw does have the CPR property. One way to spot this would be to do strictness analysis and its worker/wrapper transform first, and then do CPR analysis; but in fact, it is easy to anticipate the worker/wrapper split arising from strictness. We perform CPR analysis after strictness analysis but before the worker/wrapper transform. When we encounter a lambda binding whose strictness annotation indicates that it will certainly be evaluated, and whose type is a product, we attribute the bound variable with the CPR property. For example, when analysing fac, both n and m will have the CPR property, as indeed they do in the transformed program.

3.10 Update in place

Consider the following lazy definition:

```
f x = let y = x+1 in <e>
```

As it stands, a thunk will be allocated for y; if and when y is evaluated by <e>, the thunk will be entered and it will in turn call the function +; the latter will compute its Int result including allocating it on the heap, and return to the thunk; the thunk will finally overwrite itself with an indirection to the newly-allocated Int.

Once the definition of + has been inlined, we get:

```
f x = let y = case x of
                 MkInt i -> MkInt (i + # 1)
      in <e>
```

A thunk for y will still be allocated, *but, if and when* y *is evaluated, the allocation of the* Int *result is performed by the thunk's own code*. So now we have the opportunity to allocate the Int result in the thunk itself, instead of allocating it separately and indirecting the thunk to the Int; that is, we can perform update in place on the thunk. To make update-in-place possible, we merely need to make sure that the thunk is big enough to hold the result, which can be done on a case-by-case basis.

Why is all this relevant to CPR analysis? Because inlining the wrapper of a CPR function will expose the allocation of the result at the original call site, where it can sometimes now be performed by update-in-place. So CPR analysis makes update-in-place applicable more often.

3.11 Allocation and space safety

A good optimisation should have the following properties:

- It should not increase, and often decrease, the amount of heap allocation. Haskell programs allocate like crazy, and memory access is increasingly expensive, so heap allocation is a very important metric.

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

- It should preserve *space safety*; that is, it should not increase the program's instantaneous memory requirements (Appel, 1992). In particular, a tail recursive function should not be transformed into a non-tail-recursive one, otherwise a loop that previously executed in constant stack space may take stack space proportional to the number of iterations.

The purpose of CPR analysis is to guarantee these properties. More concretely, consider again the general form of the CPR worker/wrapper transform from section 2.1. The original function:

```
f :: Int -> (Int, Int)
f x = <e>
```

is transformed to:

```
f :: Int -> (Int, Int)
f x = case fw x of
        (# r1,r2 #) -> (r1,r2)

fw :: Int -> (# Int, Int #)
fw x = case <e>  of
         (r1,r2) -> (# r1,r2 #)
```

The idea is that *if* <e> *has the CPR property, then the* case *expression is guaranteed to cancel with a construction in* <e>, using standard simplification rules. We have seen many examples of this cancellation in earlier sections, and it is key to our claims.

If <e> has the CPR property then overall heap allocation is guaranteed not to increase because, at worst, the result allocations are eliminated (by cancellation with the case) and are replaced by the result allocation in the wrapper. This guarantee is weakened slightly by the compromise described in section 3.5, where we effectively move the allocation of a constant inside a loop. If we want an absolute guarantee, we can disable the compromise, and accept fewer functions having the CPR property; in practice we find the practical results are better if we use the compromise.

Space safety and tail recursion are another matter. At first sight it might seem that the worker/wrapper transform destroys the tail-recursive property. It looks as if the transformed f is no longer tail recursive, even if the original definition was: the call to fw is wrapped in a case, and <e> is wrapped in a second case. Nevertheless, the function g in section 3.3 started with a tail-recursive g and ended up with a worker, gw that was also tail recursive. Can we guarantee this in general?

Yes, we can. The second case, in the body of fw, moves inside <e> until it scrutinises the tail call(s) in the body of f – you can see that happening as we transformed gw in the previous section. Now, any such tail call must be to a function with the CPR property, perhaps f itself or perhaps some other function with the CPR property. How do we know that? Because that is the whole purpose of the analysis we describe in section 4. If the case *does* scrutinise a call of a function with the CPR property, we can inline that function's wrapper, and the two cases

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

(one from the call site, the other from the inlined wrapper) are then guaranteed to cancel out, exactly as we saw in the case of `gw`.

So while the first call to `f` may indeed be wrapped in a `case`, from `f`'s wrapper, any subsequent tail calls from the original program are guaranteed to be tail calls in the transformed program. Admittedly we do not have a formal proof of this property, but we are confident of its truth. Notice that the space-safety property depends crucially on the conservative nature of the analysis; a type-driven transformation would not be space-safe.

This concludes our discussion of the issues we came across when scaling up our initial idea to a full-blown compiler. We now turn to a more formal account of the analysis itself.

4 CPR analysis

The language we use to describe the analysis is based on Core, the intermediate language of GHC. Core is an explicitly-typed, polymorphic lambda calculus, based closely on System F (Girard, 1990). As such, it includes explicit type abstractions and applications. However, the form of the analysis is such that the type annotations play no part, and we omit them to simplify the presentation. In particular, the abstract interpretation yields constant functions, so issues of analysing polymorphic functions do not arise.

Core expressions are defined as follows:

$$\begin{aligned} e ::= {} & x \mid \backslash x \texttt{ -> } e \mid e_1\, e_2 \mid C\ e_1 \ldots e_n \mid \\ & \texttt{case}\ e\ \texttt{as}\ x\ \texttt{of}\ \{C_i\ x_{1_i} \ldots x_{n_i} \texttt{ -> } e_i\}_{i=1}^{m} \mid \\ & \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e \end{aligned}$$

Literal values of base types, such as unboxed integers `Int#`, are written as nullary constructors C. All constructors are saturated; that is, they are applied to all their arguments.

The `let` expression binds a mutually-recursive set of variables.

The form of the `case` expression is a little unusual. Default alternatives use the nullary pseudo-constructor `Default` and the binding e `as` x makes the selector value available in each alternative. For example, instead of the more conventional `case` e `of` $\ldots x$ `->` e_d we write `case` e `as` x `of` $\ldots$ `Default ->` e_d. This binding for x, which covers all the alternatives, not only the `Default` one, corresponds to the fact that e returns a value that is available in each alternative.

4.1 Abstract domains

Our abstract interpretation $[\![_]\!]_$ maps an expression e of type t in an environment ρ to an abstract value $[\![e]\!]\rho$, drawn from the abstract domain A_t. All abstract domains consist of either one or two points.

Type expressions are defined as follows:

$$t ::= B \mid a \mid t_1 \rightarrow t_2 \mid T\ t_1 \ldots t_k$$

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

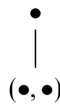
where T is a type constructor, $k \geqslant 0$, and a is a representative type variable. Base types, B, are such things as unboxed integers `Int#`.

We also assume a *collection* of type constructor definitions, each of the form:

$$T\ a_1 \dots a_k = C_1\ t_{1_1} \dots t_{n_1} \mid \dots \mid C_m\ t_{1_m} \dots t_{n_m}$$

Altogether the definitions are potentially mutually recursive.

Suppose for example that e has type `(Int,Int)`. The abstract domain $A_{(\texttt{Int,Int})}$ is a two-point lattice containing the values $(\bullet, \bullet)$ and $\bullet$. If $[\![e]\!]\rho = (\bullet, \bullet)$, then e is certain to return a constructed pair. On the other hand, if $[\![e]\!]\rho = \bullet$, then e may or may not return a constructed pair. We order this two-point domain with $(\bullet, \bullet) \sqsubseteq \bullet$, as in the following Hasse diagram:



Notice that the most uninformative value, $\bullet$, is the top of the domain, not the bottom. This is the same choice as is made by abstract interpretation for strictness analysis, which uses $\perp$ to indicate definite divergence and $\top$ to indicate lack of knowledge, and for the same reason: the least fixed point of a recursive definition is the one we want (sections 3.3 and 3.4).

In general the abstract domain A_t for a type t is defined as follows. If t is a flat base type then its values cannot be products. The only possible abstract value is $\bullet$:

$$A_B = \{\bullet\}$$

An unusual feature of CPR analysis is that the abstract value of a function is a constant function, which ignores the value of its argument:

$$A_{t_1 \to t_2} = \{\bullet\} \to A_{t_2}$$

More will be said on this in section 4.4. Observe that $A_{t_1 \to t_2} \cong A_{t_2}$ so the abstract domains of functions also consist of only one or two points.

If T constructs a product type – that is, its definition is of a form with no alternates and one or more components:

$$T\ a_1 \dots a_k = C\ t_1 \dots t_n$$

where $n \geqslant 1$, then for any types $t'_1 \dots t'_k$:

$$A_{T t'_1 \dots t'_k} = \{\bullet, \bullet^n\}$$

We write $\bullet^n$ as shorthand for the n-tuple $(\bullet, \dots, \bullet)$. Values of a product type may or may not be newly constructed so the corresponding abstract domain has only two values, $\bullet$ and $\bullet^n$. For reasons explained in section 4.3, we do not analyse within nested products, so the types $t_1 \dots t_n$ in the definition above are irrelevant.

Note that $\bullet^1 = (\bullet) \neq \bullet$. This case arises for unary products with a single field, such as the `Int` type (section 2.2):

```
data Int = MkInt Int#
```

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

If an expression e of type `Int` has abstract value $(\bullet)$, then e does construct the `MkInt` box.

If T constructs a sum type – that is, its definition is of a form with two or more alternates:

$$T\ a_1 \ldots a_k = C_1\ t_{1_1} \ldots t_{n_1}\ | \ldots |\ C_m\ t_{1_m} \ldots t_{n_m}$$

where $m \geqslant 2$, then for any types $t'_1 \ldots t'_k$:

$$A_{T t'_1 \ldots t'_k} = \{\bullet\}$$

immediately because it is not a product type. Recall from Section 3.2 that we do not attempt to treat sum types.

Since we do not analyse nested products and we abstract functions to constant functions, polymorphic and recursive data types can be dealt with simply, because our only interest is whether the *top-level* type constructor is certain to be a product. Thus we can complete the definition of the abstract domains with:

$$A_a = \{\bullet\}$$

For example, the product type:

```
Pair a b = MkPair a b
```

has corresponding abstract domain $\{\bullet, (\bullet, \bullet)\}$ irrespective of the types that may be substituted for `a` and `b`. Furthermore, the recursive definition:

```
Stream a = MkStream a (Stream a)
```

also corresponds to the abstract domain $\{\bullet, (\bullet, \bullet)\}$. The recursive part is not an issue because the analysis does not look inside product components. Recursive types usually have a sum as the top-level constructor so the corresponding abstract domain is most often just the singleton $\{\bullet\}$.

In summary, the abstract domain definitions are as follows:

$$\begin{array}{rcll} A_B & = & \{\bullet\} & \\ A_a & = & \{\bullet\} & \\ A_{t_1 \to t_2} & = & \{\bullet\} \to A_{t_2} & \\ A_{T t'_1 \ldots t'_k} & = & \{\bullet, \bullet^n\} & \text{where } T \text{ constructs an } n\text{-ary product type} \\ A_{T t'_1 \ldots t'_k} & = & \{\bullet\} & \text{where } T \text{ constructs a sum type} \end{array}$$

As we have seen, each A_t is a simple domain with either one or two points, regardless of t; so we could denote the elements of each two-point A_t simply as $\{\top, \bot\}$. When it is more convenient we do use the generic $\bot$ symbol, but we find the more explicit denotations helpful to our understanding.

4.2 *Abstract interpretation*

The analysis is a standard, straightforward abstract interpretation (Hankin & Abramsky, 1986). First, we assume the existence of a table, built during an earlier compiler pass and indicating whether constructors are of a product type. This is

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

conveniently represented as a partial function mapping constructor function names to boolean values. C is a product constructor if and only if $\phi\, C$ is true:

$$\phi :: \text{ConstrName} \rightarrow \text{Bool}$$

The abstract interpretation is defined in the usual way as a non-standard semantics over the abstract domains. First we define an environment to be a type-respecting partial function from identifiers to abstract values:

$$A = \bigcup\nolimits_{t \in Type} A_t$$
$$\rho : \text{Env} = \text{Ide} \rightarrow A$$

The abstract semantic function $[\![_]\!] : \text{Expr} \rightarrow \text{Env} \rightarrow A$ is defined as follows:

$$[\![x]\!]\rho = \rho\, x$$
$$[\![\backslash x \texttt{ -> } e]\!]\rho = \lambda \bullet . [\![e]\!]\rho[x \mapsto \bullet]$$
$$[\![e_1\, e_2]\!]\rho = [\![e_1]\!]\rho \;\; \bullet$$
$$[\![C e_1 \ldots e_n]\!]\rho = \textbf{if } \phi\, C \textbf{ then } \bullet^n \textbf{ else } \bullet$$
$$[\![\texttt{let } \{x_i = e_i\}_{i=1}^n \texttt{ in } e]\!]\rho = [\![e]\!](\mathit{fix} \lambda \rho'.\rho[x_i \mapsto [\![e_i]\!]\rho']_{i=1}^n)$$
$$[\![\texttt{case } e \texttt{ as } x \texttt{ of } \{C_i\ x_{1_i} \ldots x_{n_i} \texttt{ -> } e_i\}_{i=1}^m]\!]\rho = \bigsqcup_{i=1}^m [\![e_i]\!]\rho[x_j \mapsto \bullet]_{j=1}^{n_i}[x \mapsto [\![e]\!]\rho]$$

As a simple example, $[\![\texttt{MkInt 3}]\!]\rho = (\bullet)$, indicating a newly constructed 1-product. On the other hand, $[\![\texttt{True}]\!]\rho = \bullet$ since `Bool` is a sum type.

We can guarantee that the abstract value of the scrutinee in a `case` expression must be either $\bullet$ or $\bullet^n$. Either way, the appropriate value for the newly bound variables $x_{1_i}, \ldots x_{m_i}$ in the alternatives is always $\bullet$.

The abstract interpretation is largely standard, but there are two particular features that are worthy of further comment. First, the analysis does not look at the components of products, stopping at the top-level constructor. Second, functions are abstracted to constant functions. There are sound justifications for these choices that go beyond our desire for a simple implementation. We discuss them in turn in sections 4.3 and 4.4.

4.3 Not unboxing product components

It is a straightforward matter to define an abstract interpretation that also analyses the *components* of a product for the CPR property but it is far from clear how to make use of such nested information in a general way. In fact, if we try to take advantage of the knowledge that a component of a constructed product is also a constructed product, the worker/wrapper transformation is no longer semantically correct. Consider applying a nested CPR analysis to the following example functions:

```
g :: Bool -> Int
g x = if x then MkInt 0 else MkInt 1

f :: Bool -> (Bool, Int)
f x = (x, g x)
```

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

Now g returns a newly constructed integer so, if we were analysing for nested products, the abstract value of f's result would be $(\bullet, (\bullet))$ indicating that the second component of the result is the integer newly constructed by g.

The obvious way to make use of nested information is to 'flatten' the result of the worker function. That is, f's worker would return an unboxed tuple *and* the second component would be an unboxed integer:

```
fw :: Bool -> (# Bool, Int# #)
fw x = case g x of MkInt z -> (# x, z #)
```

Unfortunately this transformation is invalid because it now forces the evaluation of g x. Before the worker/wrapper transformation, $\mathtt{f}\,\bot = (\bot, \mathtt{g}\,\bot)$ but afterwards $\mathtt{f}\bot = \bot$, so in general it is not safe to use nested product information in this way because we risk losing laziness.

On the other hand, if we can be certain that the nested components returning constructed products do not diverge then the 'flattening' transformation is both safe and advantageous. The principal special case is where the nested components have explicit constructors. A function like:

```
f :: (Int, Int) -> (Int, Int)
f (MkInt x, y) = (MkInt (x +# 1), y)
```

will split to give a worker like:

```
f :: (Int, Int) -> (Int, Int)
f (MkInt x, y) = case fw x y of
                   (# x', y #) -> (MkInt x', y)

fw :: Int# -> Int -> (# Int#, Int #)
fw x y = (# (x +# 1), y #)
```

While this looks like a promising avenue to explore, our current CPR analysis does not consider any nested components.

In a strict language there would be no such problem, and there is no technical difficulty with enriching the abstract domains appropriately; indeed, that is partly why we chose our '•' notation rather than using simply $\top$ and $\bot$.

4.4 Constant functions

The abstract interpretation ignores the CPR property of the *arguments* to functions, thus yielding constant functions in the analysis. While it is not difficult, and perhaps more natural, to account for the arguments, it is not clear whether such information gives us any advantage. Suppose our analysis was of this form. Then a function like:

```
f x = if e then x else (3,5)
```

would abstract to

$$\mathtt{f}'\alpha = \alpha \sqcup (\bullet, \bullet) = \alpha$$

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

so the result of `f` would certainly be a newly constructed pair only if its argument was too. Nevertheless, *if* `f` *were applied only to newly-constructed pairs* – so a typical call site is `f (p,q)` – then we could pass the components of the pair to `f`'s worker, and construct a pair from the result.

In general, though, we cannot see all of a function's call sites, so we would have to compile *two versions* of `f` – one which expects a boxed argument and another which expects an unboxed argument. We would then need to inspect each of `f`'s call sites to decide which version to call. We have not, so far, tried such multi-version approaches, and our gut feel is that the law of diminishing returns will apply to such approaches.

As an aside, it is interesting to observe that if we did pursue this approach, even functions with multiple arguments still only require at most two versions. Take for example a function with three arguments:

```
f :: Bool -> Int -> Int -> Int
f True  y z = y
f False y z = z
```

Now, clearly we can only be *certain* that `f` returns a newly constructed product if *both* `y` and `z` are themselves newly constructed products. If either is not, we must use the vanilla version of `f`.

In section 3.9 we discussed the situation where the analysis assumes the CPR property for product arguments with strict demand. This special case requires a variant on the second clause defining the abstract interpretation function above. Supposing $\backslash x \texttt{->} e$ is strict and expects an n-tuple product argument, the alternative clause would be:

$$[\![\backslash x \texttt{->} e]\!]\rho = \lambda \bullet . [\![e]\!]\rho[x \mapsto \bullet^n]$$

Note that the analysis will still deliver a constant function in such cases.

A direct consequence of abstracting to constant functions is that any issues relating to the analysis of polymorphic functions disappear. Since A_a is just $\{\bullet\}$, it is irrelevant whether a polymorphic function might be called at a product type, because the abstract functions ignore their arguments.

4.5 Recursive definitions

One of the serendipitous aspects of abstracting to constant functions is that the calculation of fixed points in `let` expressions is greatly simplified. In particular, for directly recursive function definitions a single iteration of the Kleene process is sufficient. That is, in the calculation of:

$$\mathit{fix} \lambda \rho' . \rho[x \mapsto [\![e]\!]\rho']$$

we need only define $\rho_1 = \rho[x \mapsto \bot]$ and we immediately have the least fixed point, $\rho[x \mapsto [\![e]\!]\rho_1]$.

More generally, in the case of mutually recursive definitions, the number of iterations is bounded above by the length of the cycle of recursive calls. That is, for

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

the expression $\texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e$:

$$\bigsqcup_{i=0}^{n+1} (\lambda\rho'.\rho[x_1 \mapsto [\![e_1]\!]\rho', \ldots x_n \mapsto [\![e_n]\!]\rho'])^i(\bot)$$

It is clear that the fixed points of a collection of n mutual recursive constant function definitions can be found in at most n iterations of the Kleene limit construction. In $\texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e$ observe that:

- The longest possible static chain of recursive calls is n, that is, a simple cycle.
- At least one of the definitions, say $x_k = e_k$ will have an escape from the recursive loop. If not, all functions will be $\bot$ and we are done. Since the abstract domains are at most two points, the first Kleene iteration will fix any of the functions whose escape from the recursive cycle does not have the CPR property. Suppose $x_k = e_k$ is such a case. The fixed point calculation will proceed:

$$\begin{aligned} \rho_1 &= \rho[x_1 \mapsto \bot, \ldots x_n \mapsto \bot] \\ \rho_2 &= \rho[x_1 \mapsto [\![e_1]\!]\rho_1, \ldots x_k \mapsto \top, \ldots x_n \mapsto [\![e_n]\!]\rho_1] \end{aligned}$$

  If there is no such e_k then the abstract value of all the x_i will remain $\bot$.
- Having thus broken the cycle, at most $(n-1)$ further iterations will be required to propagate the abstract value of x_k through the remaining recursive definitions.

In many cases, fewer iterations may be required, but it is unlikely that using a more complex test of completion would be worthwhile. Furthermore, assuming strongly-connected components are extracted in an earlier compiler phase, `let` expressions will contain minimal collections of mutually recursive definitions. Such collections will usually be small and hence so will be the number of iterations.

4.6 Correctness

For our analysis to be *safe* we must ensure that it never incorrectly infers that a function has the CPR property. Otherwise the worker/wrapper transformation could be detrimental, as discussed in section 2.3. Of course, the analysis may be (and is) *conservative* in that not every CPR property is discovered – there is a natural trade-off between the simplicity and effectiveness of static analyses.

What conditions do we expect to hold for our analysis to be considered safe? First and most obviously, that if $[\![e]\!]\rho = \bullet^n$ then e evaluates to a product with arity n. From the discussion in section 4.3 we also need to guarantee that CPR information does not come from within nested products. Furthermore, as explained in section 4.4 we also wish to guarantee that the analysis does not make use of CPR information about the arguments to functions.

Recall that the whole purpose of the exercise is to deduce an *intensional* property of a function, not an *extensional* property. So we should not expect to be able to prove the analysis sound with respect to a standard denotational semantics. (Strictness analysis is interesting because a standard denotational semantics *is* enough to

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

deduce, at least informally, an intensional property – namely that the function will evaluate its argument – but we are not so lucky here.)

We therefore extend a standard denotational semantics for our language to include *tags* to indicate whether a value is a product that has been constructed in the current scope. Taking the standard semantic domain of each type t to be D_t, we define a non-standard extension:

$$D'_t \;=\; D_t \times Bool$$

The idea is that $\langle \varepsilon, True\rangle$ indicates that ε is a product value that has been constructed in the current scope and $\langle \varepsilon, False\rangle$ indicates that it is not. The other semantic domains are defined in the usual way. Note that there may be tags on the components of values of constructed types.

As a more lightweight notation we will write $\underline{\varepsilon}$ for $\langle \varepsilon, True\rangle$ and (slightly ambiguously) ε for $\langle \varepsilon, False\rangle$. When we leave a scope we may need to delete some of the tags, for which we will use the operator $\overline{\langle \varepsilon, b\rangle} = \langle \varepsilon, False\rangle$.

The tagged semantics is easy to define. All the clauses except those for constructors and lambda abstractions are quite standard (Peyton Jones & Launchbury, 1991).

$$\sigma : \text{Ide} \rightarrow \bigcup_{t \in Type} D_t$$

$$\mathscr{E}[\![\backslash x \text{ -> } e]\!]\sigma = \lambda\varepsilon.\mathscr{E}[\![e]\!]\sigma[x \mapsto \overline{\varepsilon}]$$

$$\mathscr{E}[\![C]\!]\sigma = \textbf{if } \phi\, C \textbf{ then } \lambda\varepsilon_1 \ldots \varepsilon_n.\underline{\langle C, \overline{\varepsilon_1}, \ldots, \overline{\varepsilon_n}\rangle} \textbf{ else } \lambda\varepsilon_1 \ldots \varepsilon_n.\langle C, \overline{\varepsilon_1}, \ldots, \overline{\varepsilon_n}\rangle$$

The tagging is simple – when a product constructor C is applied, the resulting value is marked accordingly. Everything else is left unmarked, that is, it carries a *False* tag. Furthermore, any tags on the product's components are deleted. This reflects our intention that no nested products will be considered. The clause for $\backslash x$ `->` e deletes tags from incoming values, thus ensuring that only *locally* constructed product values will be tagged.

The main result we seek is an agreement between the value of an expression under the tagged semantics and its abstract value according to the CPR analysis. To that end, we inductively define a type-indexed set of relations $\boldsymbol{sat}_t$ as follows. We simplify the notation by leaving out the type constraints, on the general understanding that variables are universally quantified and that the definitions respect types.

Definition 4.1

- $\varepsilon\ \boldsymbol{sat}\ \bullet$
- $\underline{\langle C, \varepsilon_1, \ldots, \varepsilon_n\rangle}\ \boldsymbol{sat}\ \bullet^n$
- If $\varepsilon\ \varepsilon'\ \boldsymbol{sat}\ \alpha$ for all ε' then $\varepsilon\ \boldsymbol{sat}\ \lambda\bullet.\alpha$

The relation can be naturally extended to environments: $\sigma\ \boldsymbol{sat}\ \rho$ if for all $(x \mapsto \alpha)$ in ρ, there is a corresponding binding $(x \mapsto \varepsilon)$ in σ such that $\varepsilon\ \boldsymbol{sat}\ \alpha$.
Finally, we are in a position to demonstrate the soundness of CPR analysis with respect to the tagged semantics:

Proposition 4.2
For all expressions e, environments σ and abstract environments ρ, if $\sigma\ \boldsymbol{sat}\ \rho$ then $\mathscr{E}[\![e]\!]\sigma\ \boldsymbol{sat}\ [\![e]\!]\rho$.

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

Proof
A straightforward structural induction on e. □

As with strictness analysis, there remains only an informal connection between our extended denotational semantics and the actual operational effects we seek, but we still find the denotational model quite convincing, and it has the merit of simplicity and tractability.

5 Implementation

The CPR analysis has been available in the GHC Haskell compiler since version 4.04, released in July 1999. The analysis (and associated transformation) pass is performed whenever programs are compiled with the optimisation flag (`-O`). In this section we briefly describe the implementation. The following section describes experimental evaluations of its effectiveness.

GHC is organised as a front end which, after parsing and typechecking, translates a Haskell program to a simple internal functional language called Core, for details see Peyton Jones & Santos (1998). A number of optimisation passes follow; each converts a Core program to a more optimised Core program. Finally, the GHC back end generates code for the optimised Core program. Almost all optimisations performed by GHC are carried out as Core-to-Core transformations.

This framework is readily extended with CPR analysis, simply by adding the CPR analysis pass in between strictness analysis and the worker/wrapper transformation.

5.1 The analysis

The CPR analysis is performed in a single pass over the code (except for mutually recursive bindings, where we iterate to a fixed point, see section 4.5). Since all Core expressions carry their type, as inferred by the type checking in the front end, it is trivial to discover if a constructor application produces a product type.

We abstract Core expressions to members of the following abstract domain:

```
data AbsVal = Top | Tuple Int | Fun AbsVal | Bot
```

`Top` corresponds to $\bullet$, `Tuple` n corresponds to $\bullet^n$, `Fun AbsVal` corresponds to $\lambda\bullet.AbsVal$ and `Bot` is the least element of the lattice. Having an explicit `Bot` element is not strictly necessary, since it is always expressible in terms of the other elements of the lattice (depending on the expression's type), but it is convenient for our implementation.

5.2 Separate compilation

An important property of our approach is that it is fully compatible with separate compilation: each function is analysed and transformed, independently of its call sites.

Whenever it compiles a module, `M`, GHC writes a *interface file*, `M.hi`, that contains GHC-specific information – such as arity, strictness, specialisations, and possibly

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

even the complete definition – about each of `M`'s exports. When compiling any module that imports `M`, GHC consults `M.hi` to find out about `M`'s implementation.

Given this framework, all we need to do is to write the CPR annotations on exported functions into the interface file. The CPR information is now available to any importing module just like the other GHC-specific information about the function.

6 Results

To evaluate the effectiveness of the CPR analysis and Core transformation we performed a number of experiments, which we describe below. We compare the execution of programs compiled without the CPR transformation to those compiled with it (in all other respects the compiler is identical). In both cases the programs are linked against libraries which have been compiled in the corresponding manner. For the tests we used the current stable version of GHC, 5.04.

The test programs used are from the `nofib` suite maintained by the GHC developers (Partain, 1992). All the tests were run on a lightly loaded, 600 Mhz Pentium III computer running GNU/Linux, with 256Mb of memory. All run time options were kept at the `nofib` defaults.

6.1 Static counts

Table 1 shows some statistics we gathered for all programs in the `nofib` suite. Of around 36,000 let or letrec bindings (both top level and otherwise), some 8,750 are function definitions; of these some 15% return a product type and have sufficient lambdas at the head of their definition. These are the functions which are eligible for the CPR transformation. Of these eligible functions our CPR pass discovers that about 70% have the constructed result property.

We also show the distribution of sizes of the returned unboxed tuple for all functions with the CPR property. As one might expect, small sizes dominate. In our implementation, if a function returns more results than there are registers, the extra results are returned on the stack. If the caller then boxes the result after all, this may result in extra memory traffic: the callee pushes the results onto the stack, and the caller copies them to the heap. The obvious thing to do is to experiment with a cut-off size for functions to have the CPR property. We have not done so yet.

Finally, we note that the average number of bindings in a mutually recursive group is only 1.58. As discussed in section 4.5, we take advantage of this to simplify detection of fixed points.

6.2 Dynamic counts

Many of the programs in the `nofib` suite have run-times which are too short to measure reliably. Each total run-time is the sum of three numbers (initialise time, program time, garbage collection time) which have each been rounded to 2 decimal places. We have, therefore, run each program 20 times with and without CPR

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

Table 1. *NoFib statistics*

Bindings	Top level	Non Top level	All
All	19,025	17,121	36,146
Function	6,514	2,236	8,750
Manifest function with product result	1,089	198	1,287
CPR function	807	100	907
Length of CPR function result type			
1	217	31	248
2	406	47	453
3	111	9	120
$\geqslant 4$	73	13	86
Non-recursive	18,024	16,599	34,263
Recursive	1,001	522	1,523
Average bindings in a recursive group	1.78	1.19	1.58

optimisation and compared the best run times.[5] We report programs which run for longer than one second without the CPR optimisation. Note that allocations are the same for every run, so allocation numbers are accurate.

To give us more confidence in our run-time results we used *cachegrind*, part of the *valgrind* (Seward, 2002) toolset. Cachegrind intercepts every instruction from a running executable and simulates the behaviour of the processor's cache. Amongst a detailed report we can see the number of executed instructions, the number of data reads/writes and the number of cache miss reads/writes.

Table 2 compares the run times, heap allocations, total instructions and (level 2) cache read/write misses measured for the programs in our test suite. In all, the `nofib` suite currently consists of 75 programs. As explained above, we limit ourselves to programs running for more than a second. This gives us a test suite of 34 programs; of which 4 are from the 'imaginary' section (`exp3_8`, `integrate`, `paraffins`, `rfib`), 6 are from the 'real' section (`cacheprof`, `compress`, `compress2`, `fulsom`, `hidden`, `symalg`), while the remaining 24 are from the 'spectral' section of the suite. The table shows the number of lines of Haskell code in each test. The figures in the last four columns represent the percentage change from the no-CPR run.

Ignoring the artificially inflated `rfib` (see discussion below), the run-time improvement due to CPR for the remaining programs has a geometric mean of 2.8%, with best improvement 30.8%, and worst slow-down 5.4%.

For heap allocations, the improvement had a geometric mean of 5.4%, with best improvement 61.9%, and in the worst case heap allocation increased by 17.2%.

In our experiments we have found the number of instructions and cache misses to be a good indicator of program performance. The results in Table 2 bear out (as expected) that avoiding boxing significantly improves cache behaviour.

[5] The best run time cannot overstate performance; longer times for the same program represent mainly system load.

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

Table 2. *The Effect of CPR*

Program	Lines	Run time No CPR	Run time CPR	Allocations CPR	Instr. CPR	Cache Miss read/write
atom	43	2.8s	0.4%	0.0%	-0.1%	0.1%/−0.1%
cacheprof	1723	4.0s	–3.5%	0.0%	–2.4%	–60.4%/–4.7%
circsim	392	8.3s	–1.9%	–0.6%	1.2%	−1.8%/−0.6%
comp_lab	934	1.9s	–1.0%	–0.1%	0.6%	−0.8%/−0.1%
compress	192	2.0s	–30.8%	–43.7%	–6.0%	–55.4%/–43.4%
compress2	148	1.7s	–17.0%	–17.2%	–15.8%	–16.4%/–17.6%
constraints	160	30.2s	1.6%	0.7%	1.1%	0.7%/1.4%
cryptarithm1	14	9.6s	0.0%	0.0%	0.0%	–1.1%/0.0%
event	501	1.3s	5.4%	17.2%	1.3%	1.4%/13.3%
exp3_8	17	1.3s	0.0%	0.0%	–0.2%	0.4%/0.0%
fulsom	834	3.6s	–7.8%	–3.5%	–8.8%	−18.9%/–5.7%
hidden	362	6.2s	–0.8%	–2.3%	–0.1%	−1.1%/–2.2%
integer	53	20.9s	0.1%	–3.0%	–1.0%	−0.6%/–3.0%
integrate	31	4.9s	0.0%	0.0%	0.4%	–0.1%/0.0%
life	31	2.4s	0.0%	0.0%	0.0%	0.0%/0.0%
listcompr	572	1.0s	1.0%	0.0%	0.0%	1.2%/0.0%
listcopy	577	1.2s	–0.9%	0.0%	0.0%	–0.3%/0.0%
mandel	85	2.2s	–7.3%	–9.7%	–0.6%	–14.3%/–9.7%
multiplier	335	1.0s	4.1%	9.3%	2.8%	8.5%/9.2%
para	274	5.1s	–0.8%	–3.8%	1.2%	–4.1%/–3.7%
paraffins	73	1.0s	–6.3%	0.1%	0.0%	0.0%/0.0%
power	85	8.4s	–1.8%	–0.1%	0.5%	0.7%/0.3%
primetest	100	2.4s	0.0%	–3.2%	–0.1%	0.6%/–5.9%
puzzle	137	1.5s	–0.7%	0.0%	–0.5%	0.5%/0.0%
rfib	7	2.9s	–24.1%	–99.9%	–15.1%	–93.3%/–99.9%
simple	859	5.1s	–1.0%	–2.3%	–2.1%	–0.2%/–2.2%
solid	1294	1.3s	–7.8%	–0.7%	–5.3%	–47.9%/–2.0%
sphere	289	1.2s	–2.5%	–3.2%	–0.5%	0.5%/–3.0%
symalg	831	1.1s	2.8%	–0.3%	0.0%	–1.0%/−1.0%
transform	1192	4.1s	1.2%	1.7%	1.3%	3.1%/1.8%
treejoin	50	1.2s	–2.6%	–3.5%	–4.0%	1.0%/–3.4%
typecheck	708	2.3s	0.4%	0.0%	0.0%	–1.5%/0.0%
wang	407	1.3s	–2.2%	0.0%	0.0%	0.0%/0.0%
wave4main	649	3.7s	–8.7%	–61.9%	–3.4%	–19.3%/–48.5%
Summary of results (not including rfib)						
Geometric mean improvement			2.8%	5.4%	1.3%	8.9%/4.9%
Best improvement			30.8%	61.9%	15.8%	60.4%/48.5%
Worst degradation			5.4%	17.2%	2.8%	8.5%/13.3%

At first sight it is disappointing that the CPR analysis seems to make little difference to many programs (half of the programs tested have run time and allocation changes of less than 3%). Since all programs make *some* use of integers and the CPR transformation applies to functions which return a constructed `Int` we would expect at least a modest improvement. However, closer inspection of the Core programs produced show that there are three main reasons for this:

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

- Unboxed tuples and primitive values have been supported by GHC for a long time. Many library functions have already been hand tuned to make use of these, effectively doing the CPR transformation manually.
- In Haskell 98 the arbitrary-precision `Integer` type is used for integer calculations unless the finite-precision `Int` is explicitly specified. The Integer type is a disjoint sum of either an efficient representation for values that fit in a word or a general version for arbitrary values. It turns out that many of the `nofib` benchmarks therefore do much of their work in `Integer` arithmetic; but `Integer` is not a product type and our CPR transformation does not apply.
- Many of the `nofib` programs manipulate lists, our transformation doesn't apply to types with more than one constructor (section 3.2). Further, it doesn't apply to the contents of lists since the content is polymorphic and must always be boxed.

Also, in general the CPR transformation only affects a subset of the functions in a program. To have a noticeable effect on the program's run time and heap allocation the CPR functions must be contributing a significant component of the program's execution.

An extreme case is the famous Fibonacci function, `rfib`, which we include only to confirm that the combination of strictness and CPR analysis is working as expected (we bumped up its argument so that it would run for over a second):

```
rfib :: Double -> Double
rfib n = if n <= 1 then 1
         else rfib (n-1) + rfib (n-2)
```

Without the CPR transformation each `rfib` result is built in the heap, and immediately taken apart by its caller. The CPR transformation returns the result in a register and no allocation is required for results at all. After the CPR and strictness transformations `rfib` is a function which operates entirely on the stack. As can be seen this has a dramatic effect on its memory and cache behaviour.

Any program which has some integer or floating-point calculation in its inner loop will reap similar benefits for the inner loop. In the `nofib` suite, `compress` and `wave4main` turn out to have this property, and CPR analysis gives a substantial speedup.

In two of our test programs, `event` and `multiplier`, the number of heap allocations actually *increases* significantly, for the reasons we discussed in section 3.5.

`multiplier` is a hardware simulator and much of its computation consists of applying hardware functions to streams of bits. The following function applies the `nand2` operation to two lists of bits, producing a result list of bits:

```
nand2 :: [Bit] -> [Bit] -> [Bit]
nand2 x y = zipWith f x y
  where f :: Bit -> Bit -> Bit
        f 1 1 = 0
        f _ _ = 1
```

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

If we do not apply the CPR transformation to `f`, GHC will spot that it either returns a constant 0 or a constant 1 and will always return a pointer to a boxed constant which it builds just once. The caller will simply place the pointer to the result in the output list.

However, `f` has the CPR property! The CPR transformation will convert `f` to a worker function which returns an unboxed 0 or 1 in a register. But, the caller wants to place this result in a list, and list contents must be boxed. So the caller will box the result. So now, each call to `f` causes a heap allocation, the exact opposite of what we are trying to achieve.

In `event` the increased heap allocations are due to a poor decision by GHC's full-laziness pass. The full-laziness pass lifts let-bound thunks out of sub-expressions in the hope that this will reduce recomputation. Unfortunately, in this case a good CPR transformation has given full-laziness an opportunity to apply, but its transformation has made the code worse.

In practice this does not seem to affect many programs. And in our experience the CPR transformation is usually a benefit, even if the advantage is often slight.

In another experiment, we tested the impact of the CPR transformation on GHC itself. We compiled two versions of GHC, one with the CPR transformation and one without. For each compiler we (rather confusingly!) compiled all the modules of the GHC compiler. In both cases the compile options were exactly the same.

As before, we compiled each module 20 times for each compiler and compared their best run-times and heap allocations. For the 128 modules that took the non-CPR compiler over five seconds to compile the compile-time improvement had a geometric mean of 0.9%, with best improvement 10.4%, and worst slow down 4.7%.

For heap allocations, the improvement had a geometric mean of 1.8%, with best improvement 3.0%, and in the worst case heap allocation increased by just 0.4%.

It is gratifying that CPR can improve the GHC compiler, even though it hardly does any numerical computation and most of its internal data structures have sum types.

6.3 Object sizes and compilation times

Since the CPR transformation moves code from a function definition to every call site, it may lead to larger object files. In practice, the ensuing transformations make this effect hard to determine. For each program in the `nofib` suite we compared the object sizes produced with and without CPR. We found the geometric mean of the size increases by only 0.8%, with minimum value –0.1%, and maximum value 3.7%.

In this paper, we have emphasised that the CPR analysis is both simple and can be implemented efficiently. To show that adding the CPR pass has little effect on GHC's compilation times we compare compilation times with and without the CPR pass. Note, this also includes the cost of the CPR transformation since in the latter case there will be no CPR annotations and hence no CPR driven worker/wrappering.

Again we use the programs in the `nofib` suite. We compiled every module 20 times, with and without the CPR pass, and took each module's best compile time. For the 114 modules that took over two seconds to compile without CPR we found

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

that the geometric mean of the increased compile time was only 2.3%. (Minimum increase was –9.8% and maximum increase was 19.2%).

We conclude that the CPR pass has negligible impact on the size of the resulting executable and only adds around 3% to compilation times. Considering the benefits discussed above, it is indeed a worthwhile addition to GHC.

7 Related work

Most modern languages allow a procedure to return multiple values simply by wrapping then in a data structure – C's support for returning a `struct` is a typical example. But the underlying calling convention almost invariably takes the form of call-by-reference: the `struct` is allocated on the stack, and a reference to it is (explicitly or implicitly) passed to the callee. The net effect is not as efficient as actually returning the results themselves in registers, which is the way that function parameters are passed.

A quite different approach to ours starts from the other end. Suppose that *by default all product types are represented unboxed*. In a call-by-value language the main problem this raises is that one cannot pass an unboxed value to a polymorphic function, except by using type analysis in the function to deal with the different representations that the caller might have passed. So the idea is to use a type-driven translation scheme to ensure that arguments to polymorphic functions are suitably boxed into a uniform representation. Leroy's beautiful paper describes this approach (Leroy, 1992); later, Shao elegantly refined it so that the compiler can exploit a spectrum of different representation choices (Shao, 1997).

An advantage of the type-based approach is that it is effective even for higher-order functions – that is, even if a function is passed as an argument to a higher-order function, its arguments and results can still use unboxed representations. This effect is achieved by wrapping a functional argument in an impedance-matching wrapper. There is a corresponding disadvantage too: it is possible to construct programs in which dynamically-composed impedance matchers repeatedly wrap and unwrap the same value before finally using it. Leroy describes this and other effects, before concluding that simpler, non-type-based approaches may be just as effective, or even slightly better than, type-based unboxing (Leroy, 1997).

A more fundamental problem with applying the Leroy/Shao approach to Haskell is that the basic assumption, namely representing product types unboxed by default, fails for a lazy language. Adopting an unboxed representation is not just a *representation* matter: it affects the *semantics* of the program because unboxed values must necessarily be evaluated. So we cannot simply adopt an unboxed representation by default. A further difference is that the Leroy/Shao translation is driven by types, whereas we take account of the form of the function's definition. However, their approach works uniformly for function arguments and results, whereas ours is concerned only with function results.

A popular approach to compilation of functional programs is to use *continuation-passing style* (Appel, 1992). In CPS, one never returns a result – instead, the continuation is called, passing the result(s) as parameter(s). At first sight, one might

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

think that the whole question of returning multiple results efficiently is already dealt with by argument flattening. But such is not the case: simple argument flattening is only effective when the compiler can see both the definition and all the call sites of a function, so it can consistently change the calling convention, and that is not the case for return continuations. Some more sophisticated analysis would be required.

An example of such an analysis is that of Hannan and Hicks (1998). They describe an analysis, expressed as a type system, that takes account of the form of a function's definition, and thereby guarantees (like us) never to increase allocation. They do this by distinguishing between 'compile-time types' and 'run-time types'; the former corresponds roughly to what we call a 'constructed value'. Furthermore, it is a fully higher-order analysis, so it stands some chance of doing arity raising on the result continuations of a CPS-transformed program. However, it is a somewhat-sophisticated *whole-program* analysis, which is a big disadvantage because it precludes separate compilation. We do not know of any practical implementation of this technique.

The Scheme language takes a different approach again: there is an explicit way at the language level to return multiple values (Kelsey *et al.*, 1998). Scheme's run-time typing makes it impossible to get efficient return conventions without this explicit programmer support, but even with that support multiple return values lead to non-trivial compilation issues (Ashley & Dybvig, 1994). We could do the same, and expose unboxed types to the programmer, but we prefer to keep our language simple – Haskell is already complicated enough! In fact, though, our compiler does support a switch to give direct programmer access to unboxed types, and we use this facility in writing some internal library functions.

The Spineless Tagless G-machine, the abstract machine at the heart of our compiler, originally returned *all* data values unboxed, including results of recursive or sum types (Peyton Jones, 1992). It could use this 'return-in-registers' convention uniformly without losing laziness because functions return *values* not unevaluated *thunks*. However, as we noted in section 2.3, such an approach can increase allocation in the case where the function returns an existing value. Furthermore, the return-in-registers convention turned out to greatly increase the complexity of updating thunks, switching between compiled and interpreted code, and dealing with stack underflow. We therefore moved to a default convention of returning a boxed value in the heap, recovering the advantages of returning multiple values in registers using the techniques described in this paper.

8 Conclusion

In a functional program, functions often return multiple results. We have described a simple analysis and accompanying transformation that optimises such functions, and that delivers useful performance improvements in practice.

The results are encouraging. While the performance of many programs hardly changes, a few improve dramatically. The baseline against which we compare is not a soft one, however. It is pretty easy to get good results for an optimisation applied to an otherwise-unoptimised program. It is much harder to improve programs that

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

have already been worked over by a raft of other optimisations, as is the case here. Compiler optimisations are like bullets. Each bullet is ineffective for many programs; but each gives a big payoff for a few programs whose inner loop it strikes. Good compilers simply deploy a hail of bullets, so that few programs will survive unoptimised. We believe the CPR analysis is a useful addition to the compiler's magazine.

There are several ways in which we hope to make CPR analysis more effective. We want to investigate the programs whose runtime increases even though their allocation decreases. We plan to experiment with a product-size cut-off. Another useful extension would be unboxed sums. Yet another would be a nested CPR analysis: for example, a function might be guaranteed to return a constructed product whose first component was also a constructed product.

Although our focus has been on Haskell, we believe that the ideas are applicable to call-by-value languages too, at least for direct-style compilers. (We discussed CPS-style compiling in section 7.) Even assuming that the intermediate language supports returning multiple values, the compiler must decide whether a function that returns a pair should return it heap-allocated or in registers. Our CPR analysis could guide this decision, in return offering better performance guarantees (section 3.11) than a simple type-driven approach. However the benefits may well be smaller: much of our gain comes from better handling of arithmetic, and a call-by-value language can be much more aggressive about handling integers unboxed by default. It remains to be seen whether the benefits would justify the costs.

Acknowledgements

We would like warmly to thank the following people, who helped us by commenting on a draft of this paper: Tony Hoare, Andrew Kennedy, Manuel Serrano, Zhong Shao, and Harald Søndergaard. We are particularly grateful to the three JFP referees for their detailed suggestions which led to significant improvements in the paper.

References

Ashley, J. M. and Dybvig, R. K. (1994) An efficient implementation of multiple return values in Scheme. *ACM Symposium on Lisp and Functional Programming*, Orlando, FL, pp. 140–149,

Appel, A. W. (1992) *Compiling with Continuations*. Cambridge University Press.

Girard, J.-Y. (1990) The system F of variable types: fifteen years later. In: Huet, G., editor, *Logical Foundations of Functional Programming*. Addison-Wesley.

Goubault, J. (1994) Generalized boxing, congruences and partial inlining. *First Static Analysis Symposium (SAS'94): Lecture Notes in Computer Science 864*, pp. 147–161. Namur, Belgium. Springer-Verlag.

Hankin, C. and Abramsky, S. (eds) (1986) *Abstract Interpretation of Declarative Languages*. Ellis Horwood.

Hannan, J. & Hicks, P. (1998) Higher-order arity raising. *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pp. 27–38, Baltimore, MD. ACM.

Kelsey, R., Clinger, W. and Rees, J. (eds.) (1998) The Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, **11**(1).

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press

Leroy, X. (1992) Unboxed objects and polymorphic typing. *20th ACM Symposium on Principles of Programming Languages (POPL'92)*, pp. 177–188. ACM.

Leroy, X. (1997) The effectiveness of type-based unboxing. *Workshop on Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department.

Partain, W. D. (1992) The `nofib` benchmark suite of Haskell programs. In: Launchbury, J. and Sansom, P. M. (eds.), *Functional Programming, Glasgow 1992, Workshops in Computing*, pp. 195–202. Springer-Verlag.

Peyton Jones, S. L. (1992) Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *J. Functional Program.* **2**(2), 127–202.

Peyton Jones, S. L. and Santos, A. (1998) A transformation-based optimiser for Haskell. *Sci. Comput. Program.* **32**(1–3), 3–47.

Peyton Jones, S. L. and Launchbury, J. (1991) Unboxed values as first class citizens. *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, pp. 636–666. Boston, MA. ACM.

Peyton Jones, S. L. and Marlow, S. (1998) Secrets of the Glasgow Haskell Compiler inliner. *Workshop on Implementing Declarative Languages*, Paris, France.

Peyton Jones, S. L., Reid, A., Hoare, C. A. R., Marlow, S. and Henderson, F. (1999) A semantics for imprecise exceptions. In *ACM Conference on Programming Languages Design and Implementation (PLDI'99)*, pages 25–36, Atlanta, May 1999. ACM.

Seward, J. (2002) Valgrind, an open-source memory debugger for x86-GNU/Linux. http://developer.kde.org/∼sewardj/, 2002.

Shao, Z. (1997) Flexible representation analysis. *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pp. 85–98. ACM.

https://doi.org/10.1017/S0956796803004751 Published online by Cambridge University Press